Software-Defined Adversarial Trajectory Sampling

Kashyap Thimmaraju[1]   Liron Schiff[2]   Stefan Schmid[1,3]

[1] TU Berlin, Germany   [2] GuardiCore Labs, Israel   [3] Aalborg University, Denmark

Abstract—Today's routing protocols critically rely on the assumption that the underlying hardware is trusted. Given the increasing number of attacks on network devices, and recent reports on hardware backdoors this assumption has become questionable. Indeed, with the critical role computer networks play today, the contrast between our security assumptions and reality is problematic.

This paper presents Software-Defined Adversarial Trajectory Sampling (SoftATS), an OpenFlow-based mechanism to efficiently monitor packet trajectories, also in the presence of non-cooperating or even adversarial switches or routers, e.g., containing hardware backdoors. Our approach is based on a secure, redundant and adaptive sample distribution scheme which allows us to provably detect adversarial switches or routers trying to reroute, mirror, drop, inject, or modify packets (i.e., header and/or payload). We evaluate the effectiveness of our approach in different adversarial settings, report on a proof-of-concept implementation, and provide a first evaluation of the performance overheads of such a scheme.

I. Introduction

Modern computer networks constitute a crucial infrastructure: enterprise and datacenter networks as well as the Internet in general need to provide high availability and robustness. These increasingly stringent dependability requirements stand in stark contrast to today's vulnerable routing system. In particular, while the problem of how to provide authenticity and correctness of topology propagation and route computation has been investigated intensively in the literature [1], [2], [3], [4], little is known about the vulnerabilities introduced by an unreliable or even malicious/adversarial infrastructure [5], [6], [7], [8], [9]. Indeed, it seems challenging to perform routing over an unreliable infrastructure.

This is problematic: attackers have repeatedly demonstrated their ability to compromise switches and routers [10], [11], [12], [13], [14], thousands of compromised access and core routers are being traded underground [15], networking vendors have left backdoors open [16], [17], national security agencies can bug network equipment and introduce hardware backdoors [18], hacker tools scan and eventually exploit routers with weak passwords, default settings are openly available on the Web, etc. The attack surface on network infrastructure is further exacerbated by vulnerable implementations of network protocols (e.g. CVE-2014-9295, CVE-2014-0160, CVE-2015-6325), especially in the context of more virtualized networks [19], as well as

by today's trend toward *open networks*, rendering it relatively easy for users to add switches and routers, establish links, and advertise routes. This concerns both wireless and wireline networks (e.g., powerline, overlay, or phone networks [20]).

The problem is a fundamental one: even large enterprises or national security agencies often cannot afford and do not have the expertise to develop their own trusted, high-performance network hardware. Rather, they need to rely on equipment that can be compromised during the supply chain [18], [21].

An unreliable routing system introduces several threats: for instance, wrongfully forwarded packets, i.e., packets following incorrect routes or "trajectories", may bypass security-critical components such as firewalls or intrusion detection/prevention systems (e.g., in MPLS [22]), and may be able to enter or leave security critical zones. Forwarding or mirroring packets wrongfully can also be used to violate isolation requirements in multi-tenant datacenters (e.g., forwarding patient data between two health-care providers using the same physical network [23], which is illegal [24]), to infiltrate VPNs [15] or to exfiltrate sensitive information (see e.g., Operation Aurora [25]).

While encryption may be used to mitigate some of these problems, cryptographic approaches require an additional infrastructure, and also come with overheads at runtime. This is undesirable especially in high-performance networks. Moreover, even encrypted traffic may leak sensitive information, e.g., about the times and frequency of communications.

Indeed, today, we lack good tools to verify routes in adversarial environments. For example, while traceroute and trajectory sampling tools are useful to verify routes in "reliable networks" [26], [27], and may still perform well in the context of faulty and heterogeneous networks [28], [29], they are insufficient in non-cooperative environments: a compromised switch or router may not reply with the correct information, and can also not conform to a scheme based on packet labeling or tagging [30], [31].

A. Trajectory Sampling

Trajectory Sampling (TS) is a low-overhead, direct and passive measurement method to infer packet routes. In a nutshell, in trajectory sampling, packets are sampled

according to a random distribution at different routers (usually defined by a hash over the immutable packet header space, e.g., the IP addresses and TCP ports), and sent to a *collector*, reconstructing the trajectories based on the collected samples. In general, in trajectory sampling, subsets of packets are sampled consistently: each packet is sampled *either at all routers it encounters, or at none*. TS features a number of interesting properties:

1) It provides a direct observation of the packets traversing the network. This is in contrast to indirect methods [32] that involve uncertainties with respect to the network's logical and physical state.

2) It is a passive measurement technique: it does not require, for example probe packets (which can introduce additional load and may be subject to a different forwarding behavior), nor does it require the modification of packets (e.g., tagging requires additional header space and rules at the routers). It is hence a particularly attractive approach to monitor the routes taken by packets for *high-performance networks* at a potentially low cost to forwarding.

3) It is a flexible technique that can be tuned for the networking environment. For example, the sampling rate can be adjusted on demand and the load on the collector can be adjusted accordingly. Furthermore, the distribution used for sample selection can be changed.

However, a major shortcoming is that classic TS relies on the assumption that all routers are *trusted*. There is no mechanism preventing adversarial routers from misreporting sampling information. As the sampling distribution is known to all the routers (a packet is either collected at all routers it encounters, or at none), an adversarial router could simply exploit knowledge about gaps in the sampling space to conceal misbehavior. For example, an adversarial router can drop all packets not being sampled and is guaranteed to go undetected unless the sampling distribution is changed. Hence what we need is a solution which tolerates adversarial routers, prevents exploiting holes in the sampling space, and detects forwarding based attacks such as packet drops and injections.

To prevent routers from exploiting holes in the sampling space, Lee et al. [15] propose routers to sample different packets with pairs of routers sampling the same packet. They call their scheme "secure split assignment" trajectory sampling. Although novel, and dare we say ahead of their time, the authors only present a first simulation of detecting packet drop related attacks. As we will shortly see, our scheme *Adversarial Trajectory Sampling (ATS)* develops their idea further.

B. Contributions

This paper makes the following contributions:
- We identify and model a wide range of routing attacks that can be launched by untrusted data plane

components such as *drop, injection, denial-of-service, man-in-the-middle, rerouting, reconnaissance, mirroring* and *modification* attacks.
- We observe that Software-Defined Networks (SDNs) provide an ideal framework to perform secure trajectory sampling in adversarial environments.
- We present parallelizable algorithms that leverage *secure and redundant* dynamic sampling schemes to detect routing attacks from a logically centralized controller.
- We formally prove the effectiveness of our detection algorithm by deriving the detection probabilities for different attacks.
- We present and evaluate an OpenFlow based prototype.

C. Paper Scope

In general, the goal of our approach is to empower the network operator to *detect* misbehavior, as opposed to *prevent* misbehavior. In other words, alone, our approach is unable to ensure a packet will not traverse certain network regions or reach certain destinations. However, we believe that the possibility to detect misbehavior is a strong incentive for routers and vendors to not deviate from the correct behavior. Moreover, we in this paper do not consider the orthogonal question of how a user should specify its desired and undesired routes to the network operator.

D. Organization

The remainder of this paper is organized as follows. Section II introduces our threat model. Section III and Section IV present the challenges and our proposed solution in detail. Section V analyzes the detection probabilities achieved by our approach. Section VI reports on our prototype implementation, and Section VII presents experimental results on the detection time and performance overheads under real traffic workloads. Section VIII discusses additional aspects and extensions of SoftATS. After reviewing related literature in Section IX, we conclude our work in Section X.

II. THREAT MODEL

We consider a network consisting of a set of *switches* (or for the purpose of this paper equivalently: routers), connected by a set of *links*. We consider an adversarial model where switches are untrusted, e.g., they may contain (hardware and/or software) backdoors that may be introduced by compromising the vendor's supply chain. Accordingly, we do not place any restrictions on what an adversarial switch can and cannot do. For example, an adversarial switch can fabricate and transmit any type of message, both in the data plane (e.g., duplicate packets) as well as in the control plane (e.g., report wrong samples); it can also arbitrarily deviate from the OpenFlow specification, all at the risk of being detected. However,

we assume that the network edge (e.g., the servers in the datacenter) are trusted. This assumption is necessary for the following reason: If a packet only traverses the edge, then it is **impossible** to detect whether the edge switch forwarded the packet correctly or not using our scheme.

More systematically, an adversarial switch may perform the following attacks (cf. Figure 1).

1) **Denial-of-service:** It can drop transit packets.
2) **Rerouting:** It can forward a packet to the wrong port (e.g., breaking logical isolations).
3) **Mirroring:** It can duplicate a packet, and e.g., send one to the correct and one to an incorrect port.
4) **Man-in-the-middle:** It can also delete packets, generate new packets, or modify the header or payload of packets (e.g., changing the VLAN to break isolation domains).

In fact, we observe that all attacks considered in this paper can be represented by the following two primitives:

1) **Drop:** An adversary chooses a specific source-destination pair, and drops some (or all) packets communicated between this pair.
2) **Injection:** An adversary injects new packets (or packets sent earlier to other destinations) to the network.

TABLE I
A COMPREHENSIVE OVERVIEW OF ATTACKS, THEIR RESPECTIVE PRIMITIVES, AND A SHORT DESCRIPTION OF THE ATTACK.

Attack	Attack primitive	Attack description
Denial-of-service	Drop	Switch simply drops the packet(s).
Reconnaissance	Injection	Switch simply injects a new packet.
Man-in-the-middle	Drop+Injection	Switch modifies the original packet **and** forwards it along the original path.
Man-in-the-middle proxy	Drop+Injection	Switch drops the original packet **and** forwards a modified version along a new path.
Mirror	Injection	Switch forwards the original packet **and** sends a modified version along a new path.

For instance, packet re-routing attacks, where an adversary reroutes a packet from some source to an illegitimate destination, can be modelled as a combination of two independent attacks where in the first attack the packet is dropped and in the second the packet is injected. Mirroring can be described as an injection of copied packets at some switch. A modification can be modelled as dropping and subsequently injecting modified packets (e.g., different header or payload). Packet delays can be abstracted as a drop and an injection attack.

Note that drops and injection are not only primitives resp. building blocks of more complex attacks, but can also be used as standalone attacks: frequent drops result in a denial-of-service, and injection could be used to contact Command-and-Control (C&C) servers of a botnet. Table I presents the attacks, attack primitives, and the attacks description.

There may be more than one adversarial switch, and adversarial switches may even collude: e.g., for covert switch-to-switch communication, one switch can inject a packet and the other drop it; or, one switch does not report the to-be-sampled packets of the colluding switch, or reports a non-existent packet. However, obviously and as we will see, the detection probability of our approach decreases with an increasing number of adversaries. Accordingly, we in the following, assume that the number of malicious and colluding switches is small. We argue that the assumption that only a subset of switches are malicious and collude, is reasonable in practice: for example, switches from different vendors or switches manufactured in different countries, are unlikely to collude. Indeed, leveraging heterogeneity is a key principle to improve security of networks [34], [35].

The solution proposed in this paper relies on a software controller (namely the OpenFlow controller) which securely distributes sampling rules and collects samples. Accordingly, we assume that this controller and its applications are trusted entities: for example, they are developed in-house and audited using static and dynamic program analyses—an affordable measure for large enterprises or security agencies. Moreover, we show that our algorithms are parallelizable, enabling a highly available implementation of these components.

III. THE CASE FOR AN SDN APPROACH

We first identify the fundamental challenges and requirements of *Adversarial Trajectory Sampling (ATS)*, and then describe how opportunities in SDN meet the stated requirements.

A. Requirements and Challenges

In general, we can decompose the challenges involved in designing a strong adversarial trajectory sampling scheme into several sub-challenges:

Computing and Assigning Redundant Sample Ranges. The sampling strategy of SoftATS is based on the split assignment approach by Lee et al. [15], which requires the computation and secure distribution of "good sample ranges". These ranges should be allocated

- *... independently at random:* given its own sampling ranges, a switch cannot guess with high probability the location or the sampling range of other switches.
- *... sampling space:* not all switches should sample the same packet. Instead, different switches should sample different packets.
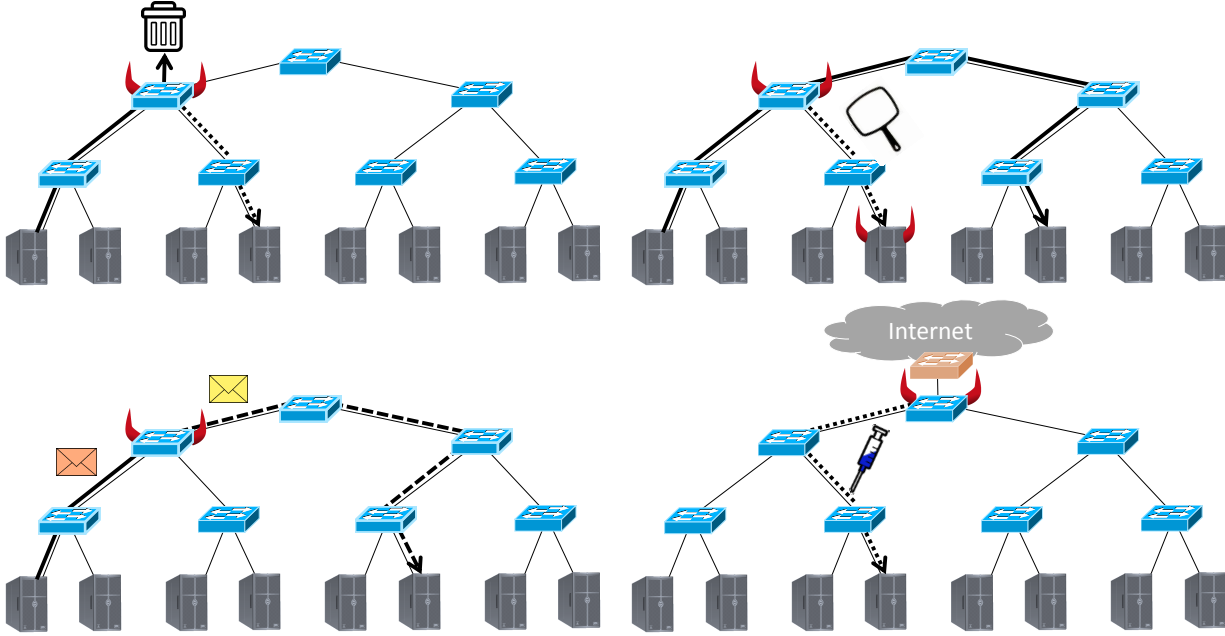
Fig. 1. Overview of possible malicious switch attacks. As an example, a Clos ("fat-tree") topology is depicted (for ease of representation, we aggregate links in this figure, see [33] for a full representation): servers are organized into racks, and are interconnected via so-called *Top-of-Rack (ToR)* switches. Racks are connected by aggregation switches to form *pods*. Finally, pods are connected by core switches, which may also connect the datacenter to the Internet. *Top left:* Denial-of-service attack resp. packet drop: instead of forwarding the packet to the server in the second rack (*dashed path*), the malicious switch drops the packet. *Top right:* The malicious switch injects a copy of the packet to the rack (*dashed path*), in addition to sending it along the regular path (*solid path*). In the rack where the packet is mirrored to, a malicious server may exfiltrate confidential information. *Bottom left:* A malicious switch modifies the packet along the route (man-in-the-middle attack). *Bottom right:* A malicious core switch injects a harmful packet to attack an internal server (an insider attack).

- ... *redundantly:* although different switches should sample different packets, there must be an overlap so that misbehaving switches can be detected by other switches.

There are different approaches to achieve such a sampling distribution. For example, for each hash range, a fixed number (e.g., two) of switches may be selected uniformly at random in the network. Alternatively, samples may be assigned in a more flexible manner, without imposing strict constraints on the number of replicas.

Securely Distributing Samples. A mechanism is needed to securely distribute samples to the switches. The samples must be encrypted to preserve confidentiality, and authenticated to preserve data integrity. However, this may be challenging when the bi-directional channel between the sample generator and switch are not connected via a dedicated network (commonly called the out-of-band network or management network). Malicious switches can drop packets to the controller that contain the samples.

Parallel Analysis. Interpolating trajectories as well as detecting anomalies requires computational resources on the collector. In order to sustain high performance workloads, the detection logic should be parallelized and distributed.

Avoiding Biases. The sampling should affect all packets with equal probability. Moreover, only an unbiased sampling can guarantee a maximal detection probability: if certain packets (e.g., packets with certain source or destination addresses) are sampled more frequently than others, this may be exploited by an attacker.

Dynamic Reassignments. Trajectory sampling, with or without split assignments, only covers a subset of packets. In order to maximize the detection probability and prevent adversaries from learning monitoring patterns, and while keeping the solution scalable, sampling distributions should be changed over time. However, this is non-trivial, as changing the assignments on the switch while traffic is sampled yields synchronization constraints [36], [37].

B. Making The Case for SDN

We find that Software-Defined Networks (SDNs) in general and OpenFlow in particular provide an ideal environment to implement an adversarial trajectory sampling for the following reasons:

1) *Programmable, logically centralized control:* The Open-Flow controller provides an ideal platform to implement the logic of the sampling and the collector: simple controller applications can be used to com-
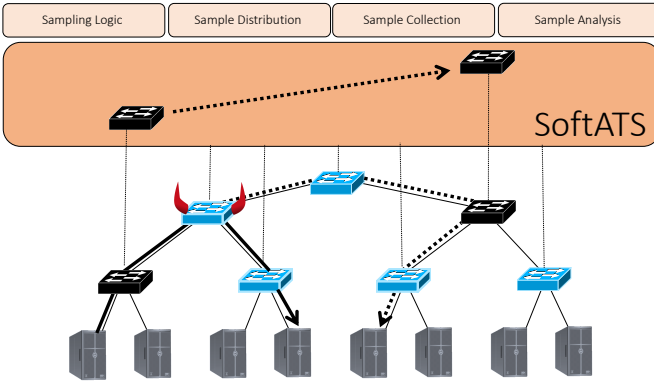
Fig. 2. Overview of SoftATS: Switches are connected to a logically centralized controller, which collects samples (*here:* from the two *black* switches) in order to observe trajectories. In this case, the samples provided by the black switches reveal that a malicious switch must have mirrored the packet (intended route indicated as *solid line*, mirrored extra path indicated as *dashed line*).

pute smart sample range distribution schemes, and to distribute, collect and analyze trajectories.

2) *Secure communication channels:* An encrypted and authenticated channel between switches (i.e., OpenFlow switches) and controller is readily supported in OpenFlow. These connections are based on TLS/SSL, include sequence numbers and could also be established in-band.

3) *Support for sampling:* An OpenFlow switch readily supports the necessary functionality for sampling. That is, a sampling range can easily be added, changed, and removed at the switch, in the form of *flow rules*. Moreover, OpenFlow switches such as Open vSwitch readily support computing (simple) hash functions [38].

4) *Support for taking into account payload:* Unlike legacy switches, OpenFlow switches can operate not only on Layer-2, but also Layer-3 and Layer-4 header fields. Accordingly, sampling can be performed on a per-packet granularity rather than a per-flow granularity. For example, sample decisions may be based on the TCP checksum field which depends on the packet payload. Moreover, some OpenFlow switches (such as NoviKit[TM]) also support sampling based on packet payloads (e.g., defined by a certain offset). As we will see, by collecting entire packets and not just headers (as readily supported by the *Packet-in* mechanism), more sophisticated attacks can be prevented.

IV. THE SOFTATS SYSTEM

In this section, we describe our Adversarial Trajectory Sampling (ATS) scheme system which provably detects a wide range of routing attacks in SDNs. In particular, we present the SoftATS architecture and our algorithms in detail.

SoftATS is designed for an OpenFlow-based network and a logically centralized (but possibly physically distributed) controller. The SoftATS controller serves as the (secure) distributor of sampling ranges, as the (secure) collector of the sampled packets, as well as the analyzer accordingly. As we will discuss later in more detail, some of this logic can easily be parallelized. SoftATS relies on the following concepts:

- **Packet Hashing Function** h**:** A hash function $h : \{0,1\}^* \rightarrow H$ which maps (parts of) the packet to some *hash domain H*. We consider two types of hashing approaches:
  1) *Header based:* The hash is computed only over the immutable header fields of the packet.
  2) *Payload dependent:* The hash also depends on the payload of the packet.
- **Hash Assignments** $A(s)$**:** Each switch s, is assigned a set of hash values $A(s)$. The switch is configured to report (sample) any packet with content that hashes to $A(s)$. The size of $A(s)$ divided by the size of the hash domain defines the **sampling ratio** of switch s, denoted by p_s. In other words:

$$p_s = \frac{|A(s)|}{|H|}$$

- **Assignment Granularity** δ**:** The set $A(s)$ can be configured in the switch using a set of configuration rules $R(s)$, each corresponding to a disjoint subset of δ hash values, e.g., through the use of a *range* or a *ternary bit pattern*. The following equality holds:

$$|R(s)| \cdot \delta = |A(s)|$$

Clearly the granularity affects the configuration space complexity.

- **Redundancy:** In order to verify the trajectory of a sampled packet in the network, it must be sampled at more than one switch (but for secrecy purposes, not all).

SoftATS is a modular framework which supports many different sample assignment strategies, leveraging SDN's flexible configuration and packet matching capabilities. Next we discuss four important elements of the adversarial sampling process in turn: (*i*) how hash assignments are applied in the switches, (*ii*) how hash assignments can be changed over time, (*iii*) how sampled packets are collected, and (*iv*) how to reconstruct the trajectories given the samples, using a highly parallelizable algorithm.

Configuring Hash Assignments for Sampling. SoftATS offers many flexibilities in terms of assignment. For example regarding the hash assignment for each switch in the network, there are different approaches, e.g.:

1) *pairwise:* For each pair of switches, $s_i, s_j \in S$, we assign a randomly selected subset of hash values $A(s_i, s_j)$. In total, each switch s_i is assigned with

the union of all subsets selected for all its pairs, i.e., $A(s_i) = \cup_{j \neq i} A(s_i, s_j)$.

2) *independent:* Each switch is assigned with a randomly selected subset of hash values, independently from other switches.

In the pairwise approach, we also ensure multiple matches (with probability 1), avoiding ineffective assignments (e.g., with local unmonitored ranges) that might be chosen in the independent assignment.

The hash function can be applied to just the packet header or the payload as well. In general, SoftATS maintains an OpenFlow session to each switch and uses it to send *Flow-mod* or *Group-mod* commands according to the switch specific hash assignment.

Packet Header Hashing. In case the hash function should be applied to the packet header, we configure the default hash based selection group with $|H|/\delta$ buckets. Among these buckets, $|R(s)| = |A(s)|/\delta$ buckets are defined to send the packet to the collector. The indices of the sampling buckets are chosen according to the assigned hash values. Note that by using weighted buckets, the total number of buckets can be linear in $|R(s)|$, e.g., replacing k consecutive dummy buckets with one dummy bucket of weight k. An example of such a configuration is shown in Listing 1

Packet Payload Hashing. In case the hash function should be applied to the packet payload, then the switch is configured to match the TCP/UDP checksum field. Matching the checksum field using OpenFlow, requires the experimenter extension and has already been prototyped by Afek et al. [39]. We emphasize that TCP/UDP checksums are only used at the switch for sampling, while deeper payload verification (hashing) can be performed at the collector. Matching the TCP/UDP checksum field alleviates the overhead of checksum computation on the switch. The flexibility of OpenFlow allows custom match fields, therefore for packets that do not include checksum fields, a custom match field and hash algorithm may be used. Concretely, assuming that the rules (subsets) $R(s)$ can be expressed by ternary patterns, s is configured with exactly $|R(s)|$ flow entries, each sending the matched packet to the collector.

In both cases, sending the sampled packets to the collector is performed by the OpenFlow action to forward to the controller as a *Packet-in*. An example of such a configuration is shown in Listing 2

Dynamic Configuration. It is useful to change the hash assignment on the switches as at any moment in time, SoftATS samples only a fraction (p_s) of the traffic. This way, over time, it becomes difficult for the adversary to avoid being detected as no information about a static sampling pattern is leaked.

However, changing the hash assignment over time is non-trivial. During the addition or removal of a hash assignment from a switch configuration, there is an uncertainty regarding the exact time at which an update takes

---

Alg. 1 Initial Pairwise Assignment of Hash Values

Require: sampling rate p, pattern size δ, total hashes H, switches S
1: $x \leftarrow p \cdot |H|/\delta \cdot (|S|-1)$ ▷ *number of hash patterns per pair of switches*
2: $assignedHashes \leftarrow 0$ ▷ *initialize the assigned hash counter*
3: $totalHashes \leftarrow x \cdot |S| \cdot (|S|-1)$ ▷ *number of hash patterns for all switches*
4: **while** $assignedHashes < totalHashes$ **do**
5:     **randomly choose** a switch-pair $(s_i, s_j)_{i \neq j} \in S$
6:     **randomly choose** a hash pattern $h \in H$
7:     **if** $|A(s_i, s_j)| < x \&\& |A(s_j, s_i)| < x$ **then**
8:        $A(s_i, s_j).add(h)$
9:        $A(s_j, s_i).add(h)$
10:        $assignedHashes \leftarrow assignedHashes + 2$
    return hash assignments A

Alg. 2 Random Permutation of Hash Assignment Values

Require: switches S, total hashes H, hash assignments A
1: **while** True **do**
2:     **randomly choose** a switch pair $(s_i, s_j)_{i \neq j} \in S$
3:     **randomly choose** a hash index $k \in \{1..x\}$
4:     **randomly choose** a hash $h \in H$
5:     $A(s_i, s_j)[k] \leftarrow h$
6:     **wait** a random period

---

effect: the communication network between the controller and the switch is inherently asynchronous, and also data structure reconfiguration times at the switches may differ [40]. To avoid false positives, SoftATS needs to temporarily suppress alerts related to the updated switch and hash values. In particular, hash values should be changed at *random times*, and involving only one (pair of) switch(es) at a time: thus, a malicious switch cannot abuse the time period of uncertainty. A simple update scheme is described in Algorithm 2.

Collecting the Samples. Reporting the samples from all switches to the collector is performed using *Packet-in*s within the encrypted and authenticated OpenFlow session, protecting them even if communication is in-band; this addresses known sample integrity issues [15]. Nevertheless, whenever possible, we suggest using out-of-band control in order to prevent sophisticated attacks, e.g., using burst analysis of the inband control channel. In case of distributed control planes, the *Packet-in*s (the samples) can be sent to the same controller which configured the sampling rules, or to another controller server depending on operator requirements.

Analyzing the Samples. At the heart of trajectory sampling lies the construction or "interpolation" of trajectories out of samples. In an SDN, the controller has a global view of the network, i.e., it knows the location of hosts in

```
group_id=1,type=select ,bucket=actions=push_vlan:0x8100 , set_field:1234−>vlan_vid ,CONTROLLER:65535 ,
bucket=actions=push_vlan:0x8100 , set_field:1235−>vlan_vid ,CONTROLLER:65535
cookie=0x1,  table=0,  priority =1,in_port=2,nw_dst=1.1.1.0/24  actions=output:3 ,group:1
```

```
cookie=0x1,  table=0,  priority =2,tcp ,chksum=1234  actions=CONTROLLER:65535 ,goto_table :1
cookie=0x2,  table=0,  priority =1,ip  actions=goto_table :1
cookie=0x3,  table=1,  priority =1,in_port=2,nw_dst=1.1.1.0/24  actions=output:3
```

the network, the switches, links between switches, and links between switches and hosts. For example, with two samples from two different switches, the controller can compute and compare the packet's trajectory using the topological information it has. This makes it ideal for trajectory computation. Based on the collected samples, anomalies such as missing or extra samples are identified, and drop or inject attacks detected accordingly. SoftATS performs this analysis in the collector following Algorithm 3, as explained next.

The sampled packets arriving at the collector, a hash of the packet, their sampling locations, and arrival times, are stored in a list ordered by time. We will refer to this list as the *history*. Packets are processed after spending *max_delay* seconds in the history, where *max_delay* is a time interval (Round Trip Time) defined to ensure that all other samples of the packet have arrived.

At the collector, the processing of a packet includes, first, constructing the trajectory of the packets, using a policy oracle. The policy oracle returns the path (trajectory) that suits the packet. Then, considering the switch hash assignments along the path, all other expected samples of the packet in the history are searched for. To detect packet modifications, hashes of the samples are compared to ensure no modifications. If a sample is missing for some switch s, we distinguish between the following two cases depending on the location of s along the path and relative to other sampling switches along the path:

1) *Path suffix only:* If the packet was sampled by s but not by switches after s, we report a drop attack.
2) *Path prefix only:* If the packet was sampled by s but not by switches before s, we report an injection attack.

While the mechanism to identify injection and dropping events are similar, the severity of, and reaction to these two events may differ. In particular, while injections may occur rather rarely "by accident", benign packet drops do. Accordingly, for drops arising individually and without statistical patterns, no alarm should be raised. To deal with the ephemeral hash value

---

Alg. 3 Detection

Require: hash assignments A, switches S, incoming event (packet-In) queue Q, network policy oracle *Policy*

1: $History \leftarrow ()$ ▷ *empty sorted list*
2: $t_0 = time()$ ▷ *current time*
3: **while** *true* **do**
4:      $timestamp, pkt, s \leftarrow Q.get()$ ▷ *blocking get*
5:      $History.add(timestamp, pkt, s)$
6:      **if** $timestamp - t_0 < RTT$ **then**
7:          *continue*
8:      **while** $History.min() < timestamp - RTT$ **do**
9:          $timestamp', pkt', s' \leftarrow History.get\_min()$
10:         $h' \leftarrow hash(pkt')$
11:         $path \leftarrow Policy.get\_path\_suffix(pkt', s')$
12:         **for** $s'' \in path$ **do**
13:             **if** $h' \in A(s'') \&\& (pkt', s'') \notin History$ **then**
14:                 *Report Drop of pkt' between (s', s'')*
15:                 *Break*
16:         $h \leftarrow hash(pkt)$
17:         $path \leftarrow Policy.get\_path\_prefix(pkt, s)$
18:         **for** $s'' \in path$ **do**
19:             **if** $h \in A(s'') \&\& (pkt, s'') \notin History$ **then**
20:                 *Report Injection of pkt between (s', s'')*
21:                 *Break*

---

assignments and avoid false positives, we introduce a grace period around dynamic updates.

Parallel Trajectory Construction. While the sampling approach implemented by SoftATS is efficient and effective, in order to scale the system further, we can easily parallelize SoftATS by executing the detection algorithm in multiple threads or by using multiple collectors that execute the detection algorithm. To increase the throughput of Algorithm 3 using multiple threads, the samples received by the collector can be evenly distributed across the threads. To increase the throughput using multiple collectors, the samples from the switches can be sent to different collectors. Such partitionings are highly scalable: sample dependencies are restricted to a single

packet.

V. ANALYSIS OF DETECTION PROBABILITY

This section presents a formal analysis of the detection probability for a single attacker in SoftATS under different attacks, and in different scenarios. We begin by describing our approach for analyzing the probability to detect various attacks based on a pairwise static assignment distribution. Following that, we describe the detection probabilities for a single packet attack, and flow-based attacks. Given that the assignment distribution is static, in the next section, we will report on prototype experiments which evaluate and extend our insights to time-varying assignment distributions.

A. General Approach and Notation

Observe that if either the original or (one of the) actual route(s) taken by a packet traverses at least two sampling points, SoftATS *guarantees* detection. This is obvious for simple routing attacks, i.e., packets which are rerouted, injected, dropped, or mirrored. However, this also holds for sophisticated attacks which involve changing packet contents. For example, consider a scenario where sampling is based on TCP checksums and assume that an attacker modifies the packet in such a way that TCP checksum remains unchanged. Even in this case, SoftATS ensures detection: with the unmodified checksum, the packet is sampled and sent entirely to the controller, which can (bit-by-bit) compare the packets, and detect modifications.

In addition to the attacked packets, whether the attacker reports the samples, or not, for the attacked packets affects the detection probability. It is better for a single attacker to report samples for packets that are dropped, and not report samples for packets that are injected. In the analysis that follows, we assume that the attacker follows this strategy with resepect to reporting samples for attacked packets.

We now focus on the pairwise assignment scheme and assume a payload packet hashing function. We consider three types of attacks: (1) single packet attacks (drop and inject), (2) dropping an entire flow, and (3) injecting a new flow. We will use the following notations:

- n: the total number of switches
- p: the same sampling ratio of the switch
- $PS(x, y)$: the number of hash range pairs involving switches from two sets of sizes x and y, i.e., $PS(x, y) = x \cdot y$
- $\widetilde{PS}(x)$: the number of pairs involving at least one switch from a set of x switches, i.e., $\widetilde{PS}(x) = x \cdot (n - x) + \binom{x}{2}$.
- $PS^-(x, y)$: the number of pairs involving switches of a set of size x but no switches from a second set of size y, i.e., $PS^-(x, y) = \widetilde{PS}(x) - PS(x, y)$

- (B, A): the number of switches along a path before (B) and after (A) the attacking switch respectively, i.e., the attack location

B. Single Packet Attack

In the single packet drop attack, the attacker drops a single packet from a flow. Similarly, in the single packet injection attack, the attacker injects a single new packet into a flow. As we will see, the probability of detecting a single packet drop attack and a single packet inject attack, is the same.

To compute the detection probability, we assume that the attacked packet belongs to a flow along a single path. The location of the attack along the path is expressed by (B, A) which stands for the number of switches along the path before and after the attacking switch respectively. Moreover we assume that all switches have the same sampling ratio, p. Therefore, each pair of switches that share an assignment contains $p/(n-1)$ of the hash space $|H|$ (independently at random from the other pairs).

Recall that a detection occurs if the packet is sampled by at least one switch before and at least one switch after the attack location. By definition, there are exactly $PS(A, B)$ pairs surrounding the attack location (B, A): we will refer to these pairs as *explicit pairs*.

However, we observe that the actual detection probability is slightly higher than what we would obtain by focusing on the explicit pairs alone. Due to a *birthday paradox*, it is probable that other pairs surround the attack location, as they happen to have the same assignment before and after the attack location. We will refer to these pairs as *implicit pairs*.

Fig. 3 visualizes the difference between explicit and implicit pairs: implicit pairs arise if B2 and X, and A1 and Y are assigned the same hash value.

To evaluate the number of implicit pairs, we need to compute the collision probability between two sets of random variables [41], where the (maximal) set sizes are given by $PS^-(B, A)$ and $PS^-(A, B)$, and the random variables can assume any of $|H|$ discrete values (or $|H|/\delta$ considering assignment granularity). Following work by Yancey [42], we use $COL(x, y) := x \cdot y / |H|$ as an approximation for the collision of two sets of random assignments of sizes x and y.

The detection probability, P_{pa}, then equals the probability that the packet is sampled by one of the pairs surrounding the attack location. Considering the number of explicit and implicit pairs, we obtain that

$$P_{pa} = 1 - \left(1 - \frac{p}{n-1}\right)^{PS(A,B) + COL\left(PS^-(B,A), PS^-(A,B)\right)}$$

(1)

We note that if the attacker does not report a sample for a dropped packet, then the probability of detection is slightly higher, as the assignments shared between the

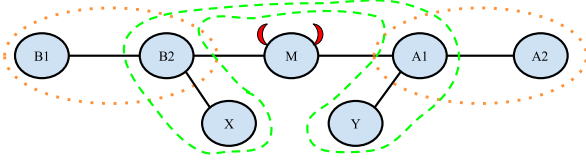|	B1	B2	A1	A2	X	Y
B1		100	200	300	700	50
B2	100		500	150	400	150
A1	200	500		600	70	400
A2	300	150	600		350	650
X	700	400	70	350		100
Y	50	150	400	650	100	

Fig. 3. Illustration of explicit and implicit pairs. An example network topology is shown below a table indicating the hash assignments for each node in the network. The table's rows and columns indicate the nodes. The assignment for a pair of nodes is given by the cell that intersects the chosen nodes. M is the attacker and its position is given by (2,2). B1 and B2, and A1 and A2, form two explicit pairs, indicated by the dotted orange lines and the orange cell in the topology and table resp. B1 and B2 share the hash value 100, and A1 and A2 share the hash value 600. B2 and X, and A1 and Y, also form explicit pairs. However, the two pairs are assigned the same hash value of 400, making them an implicit pair. The implicit pair is indicated by the dashed green line and green cell in the topology and table resp. To keep the figure simple, not all explicit pairs are shown.

attacker, and switches before and after it aid in detection. Similarly, if the attacker reports a sample for an injected packet, the chance of detection is equal to that of the drop and the attacker not reporting the sample.

C. Flow Drop Attack

Assume now a scenario where an attacker drops the entire flow. Obviously, also here, the detection probability depends on the location of the attacker (B, A) along the flow's path. Also of importance is the packet hash function used to detect this attack. In particular, hashing only the header results in the same values for all packets in the flow, and a constant detection probability P_{pa} (as for a single packet described for the single packet attack), regardless of the number of dropped packets. In contrast, hashing the packet's payload leads to more random (per packet) values, thereby making the detection of each packet an (ideally) independent experiment, each with success probability P_{pa}. Therefore, the detection probability of the entire attack describes a geometric distribution, and the expected number of dropped packets till detection is $1/P_{pa}$.

D. Flow Injection Attack

We next examine attacks injecting new flows. It is easy to see that if the injected flow's packets have uniformly distributed hash values (as assumed for the original flow), then the detection probability is like that of the drop attack. The expected number of injected packets till detection is $1/P_{pa}$ in this case.

However, if we consider a more sophisticated attack wherein the attacker may craft all of its packets to hash to the same value from the packet hash function, then the attack can either be detected with the first packet from the injected flow, or never. The initial (and static) hash assignment of the switches is a crucial factor to detect such an attack.

By dynamically configuring the switch pairs hash assignments with new random values and at random times (following a memoryless Poisson distribution), the detection probability may be improved. The update rate should be high enough to introduce many new assignments, at the same time it should keep the number of values updated low: alerts resulting from the recently updated assignments are temporarily suppressed (to avoid false positives).

The detection of the sophisticated flow inject attack will occur when the Poisson process of updates results in an assignment surrounding the attack position which includes the hash value used in the injected flow. We can distinguish between three cases, depending on whether the crafted value is assigned before the attack point, after the attack point, or not at all. The derivation of a closed-form expression for the expected detection time is difficult, as there are multiple parameters to consider, such as the duration of the attack, the update rate, the update size and the rate of traffic flowing through the path. The duration of the attack is the most important parameter: at one end of the spectrum lies the static case, at the other end of the spectrum, if the assignments change frequently between injections, the attack is equivalent to using different hash values and therefore similar to the flow drop attack.

E. Collusion

Our analysis generalizes to collusion, as follows. We assume that multiple switches collude, e.g., communicate with each other (to coordinate a later attack), and do not report each others' samples. To analyze such behavior, we can adapt our above analysis. We simply compute the detection probability considering a single attacker at a time, ignoring the other attackers along the path, effectively reducing the path length for an attacker under analysis.

VI. PROTOTYPE IMPLEMENTATION

To demonstrate the feasibility of our approach, to validate and extend the results of our formal analysis, and to investigate the potential performance overheads, we implemented a prototype of SoftATS.

We use ONOS-1.4 (implemented in Java) as our controller (collector) and implemented SoftATS as an application on top of it. We used the OpenFlow 1.3 specification implemented by ONOS for our prototype. ONOS offers the many functionalities and services required to implement SoftATS, such as multi-threading, path

service, flow rule service, device service, group service, and packet service.

Fig. 4 illustrates the architecture of our implementation. It is made up of five main components: Configuration, Hash Assignment, Sample Dispatcher, Detectors and a Trajectory Oracle. In the following we will elaborate on each of the components. After that, we will briefly describe the data plane components of SoftATS.

A. Configuration

The different parameters of SoftATS such as the sampling ratio, detector threads, pairwise assignment or independent assignment, etc. can be configured via the ONOS Command Line Interface (CLI).

B. Hash Assignment

The hash assigner executes the pairwise assignment of hash values (Algorithm 1) and utilizes the flow objective service to assign sampling flow rules to the switches when payload hashing is used. For packet header hashing, the group service is used to assign flow and group table rules. The hash assignments are then stored in a data structure that keeps track of the hash assignments for all the switches. The hash assignments are later used by the detector thread(s) to carry out the detection algorithm (Algorithm 3). For dynamic assignments, the hash permutator executes the permutation of hash assignment values (Algorithm 2) in its own thread following a Poisson distribution [43]. Its average update rate can also be configured via the ONOS CLI. It passes the new assignment to the hash assigner to have the assignment pushed to the switch(es) and have the hash assignment data structure updated.

C. Sample Dispatcher

The packet service in ONOS receives samples from the switches as *Packet-Ins*. It strips away the OpenFlow header and passes the Ethernet frame (packet) to the sample dispatcher's packet processor (FIFO) queue of SoftATS. The sample dispatcher always takes the first sample from the queue and dispatches the sample to the appropriate detector's history queue. The logic for the dispatcher and the detector threads are based on how one wants to distribute the sampling load. In our case, we divide the hash space $|H|$ evenly across the detector threads. The dispatcher simply checks which detector thread is responsible for the particular sample (according to its hash) and places the sample into the detector's history queue.

D. Detectors

Each detector thread executes the detection algorithm (Algorithm 3) and therefore has its own history queue which allows it to operate independent of the other threads. There are no dependencies on the other threads. The only concurrency requirement on the history queue is that operation on it be synchronized.
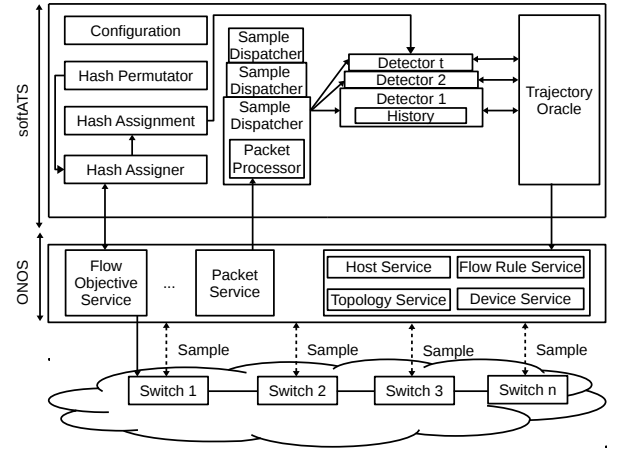

Fig. 4. A high-level overview of the SoftATS software architecture. The *Hash Assigner*, executes Algorithm 1, stores them in *Hash Assignment*, and uses the Flow Objective Service or Group Service to install sampling rules on the switches. A received sample is processed by the *Sample Dispatcher* and sent to the appropriate *Detector* thread. The *Detector* uses the *Hash Assignment*, and path information obtained from the *Trajectory Oracle* (which has a global network view) to detect whether an attack has occurred or not, and report accordingly.

E. Trajectory Oracle

The network policy oracle implemented as the trajectory oracle in our prototype, returns the path suffix and path prefix to the detector by obtaining host, device and path information from ONOS services. If an attack is detected by the detector, an alert is generated in the log messages of ONOS.

F. Data Plane

For the data plane, we use Open vSwitch (OvS) as it readily implements OpenFlow 1.3 and supports packet header hashing via *group tables*. By default, OvS, ONOS and OpenFlow do not support matching the checksum field of TCP/UDP packets in a flow rule [44]. Therefore, we use the currently supported VLAN tag field to match packets for sampling. We tag all our traffic and populate it with the value from the checksum field truncated to the tag field's size. Since the tag field is only 12 bits and the checksum field is 16 bits, we effectively reduce $|H|$ from 65535 to 4096, which we deem acceptable for our proof-of-concept prototype and experiments.

VII. Experimental Evaluation

Using our prototype and realistic traffic workloads, we have conducted extensive experiments on a Clos topology (a "fat-tree"). We first report on our experimental setup. Following that, we report on the observed detection times, and then discuss the throughput of computing trajectories in SoftATS. Finally, we study the resource overheads introduced by SoftATS.
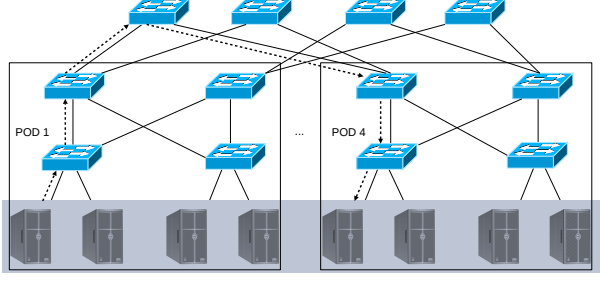
Fig. 5. Topology used for detection time, trajectory throughput and sampling overhead experiments of SoftATS. The traffic between hosts flows as per the arrowed and dotted line i.e., traffic flows from a host in Pod 1 to a host in Pod 4.

A. Setup

We consider a Clos/fat-tree topology with $k = 4$, using the Ripcord platform for Mininet [45], [46]. To generate realistic network traffic, we leverage the Lawrence Livermore Berkeley National Laboratories (LLBNL) traffic [47] traces. Using a custom script, we extract only the internal subnet-to-subnet flows from the traces, then extract the TCP checksums from those flows and replay traffic using those checksums within our topology using tcpreplay at 100 pps as shown in Fig. 5. Considering the concepts defined in Sec. IV, our default parameters for SoftATS are as follows: **Detectors** t: 1; **Packet Hashing Function** h: payload dependent; **Sampling Ratio** p_s: 0.4%; **Assignment Granularity** δ: 1; **Redundancy:** Pairwise; **Average Hash Update Rate:** 2s and **Hash Update Size:** 2. The number of hash assignments per switch and sampling ratio are chosen such that every switch in the network forms one pair with every other switch. Recall from Sec. IV, $p_s = \frac{|A(s)|}{|H|}$. Since our topology has 20 switches, we have $|A(s)| = 20$, and $|H| = 4096$, we get $p_s = \sim 0.4\%$ as the sampling ratio. In the following, we will explicitly state any changes to the default parameters and traffic.

B. Detection Time

To validate and complement our formal analysis of the mean detection time, we conducted a set of experiments studying the detection time using real network traffic. In particular, we evaluate our detection algorithm (Algorithm 3) in the presence of a single attacker, and two colluding attackers. We focus on the flow drop, and flow injection attacks for the single attacker. For the colluding attackers, we focus on the flow injection attack. In both attacks (flow drop, and flow inject), we use a static and time-varying sampling distribution for our detection algorithm.

Setup: Since the sampling ratio impacts the detection time, in addition to the the default parameters specified in Sec. VII-A, we use the following: **Sampling Ratio** p_s: 0.9% and 1.3%.

1) Single Attacker: We first describe our methodology, results and analysis for a single attacker.

Methodology: We evaluate the effectiveness of detecting the flow drop and flow injection attacks when the attacker is the aggregate switch and when the attacker is the core switch along a single path in our topology. In the flow drop, the switch drops all packets from a flow along a path. Furthermore, it reports the samples, if any, for dropped packets. In the flow inject, the switch injects a new flow along the path of an existing flow. Additionally, it does not report samples for the injected packets. Both attacks are easily emulated via OpenFlow flow rules on OvS. We count the number of packets that are sent in a flow till an alarm is raised by SoftATS and then stop. We perform 100 such trials for **each attack** and **each attacker**. We use the packet count as a metric instead of time to remain independent of the traffic rate: what are "realistic rates" depends on the context (e.g., data center vs ISP).

Results and Analysis: Fig. 6 shows the results from the experiments. The theoretical means are represented as solid colored lines. The box plots represent the empirical results: the star represents the mean, and the solid red line represents the median. First, the figure confirms our theoretical results: the empirical results are very close to the analytical mean for the detection time. However, while in theory the flow inject attack takes as long as the flow drop attack, empirically it is detected sooner. This is possible for two reasons. First our analysis of the detection probability is conservative, i.e., we do not consider the pairs between the attacker, and other benign switches along the path, that can in fact help in detection. Second, the hash assignments are not purely independent, i.e., for the chosen sampling ratios, each switch had exactly one, two and three pairs shared with every other switch in the network. This increased the number of unique pairs surrounding the attacker, thereby increasing the detection probabilities. Furthermore, we observe a lot of variance in detecting the various attacks. We suspect this is due to the non-uniform distribution of the TCP checksum field in the traffic used. Partridge et al. [48] measured the performance of the TCP checksum, and identified that for small sized packets, the distribution is skewed for the UNIX file system. Nonetheless, based on our experiments, the results work in favor of SoftATS. The sampling ratio p also improves the detection, roughly linearly: by doubling p we detect the attack in half the expected number of packets. The position of the attacker also influences the detection, i.e., it takes fewer packets to detect the malicious core switch than the malicious aggregate switch. This is due to the fact that there are more explicit and implicit pairs surrounding the core switch than the aggregate switch, hence improving detection probability. In Sec. VIII-A, we analyze the relationship between the detection probability, the path length, and the attacker's position in the path.
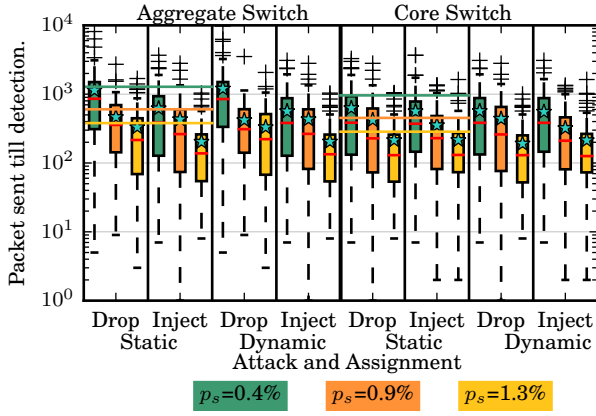
Fig. 6. Average no. of packets to detect the attacks at different positions using the pairwise assignment in the static and dynamic setting with an average update rate of 2s and an update size of 2 and varying sampling rates.
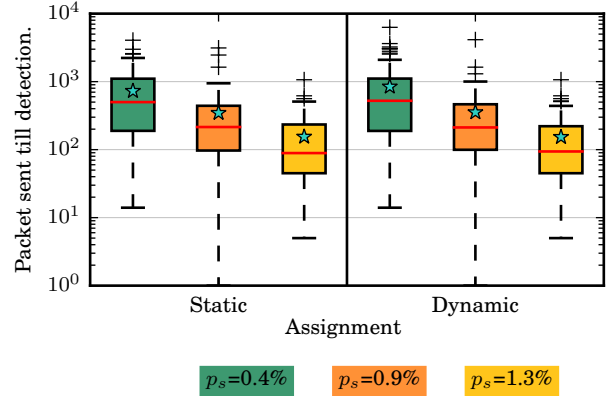

Fig. 7. Average no. of packets to detect a flow inject attack when two switches collude, using the pairwise assignment in the static and dynamic setting with an average update rate of 2s and an update size of 2 and varying sampling rates.

2) Colluding Attackers: Now we describe our methodology, results and analysis for two colluding attackers. **Methodology:** We evaluate the effectiveness of detecting the flow injection attack (analogous to mirroring) when two aggregate switches collude: the two switches collude to not report samples for all packets injected. We emulate that by modifying the specific sampling flow rules on the two switches to not report matching packets to the controller. In this attack, the benign traffic flows from Pod 1 to Pod 4 (as shown in Fig. 5. However, the injected traffic flows from Pod 1 to Pod 3. The remainder of the methodology is the same as the single attacker in Sec. VII-B1.

Results and Analysis: Fig. 7 shows the results from the collusion experiment. The figure shows that for two colluding switches not reporting samples for all injected packets, SoftATS requires less than 1000 packets to detect the attack. Naturally, we observe that with fewer benign switches, it takes more packets to detect an inject attack. Using the dynamic assignment, we observe that the values chosen do not result in significant changes to the detection compared to the static assignment.

C. Detection Throughput

Next, we study the number of trajectories per second which can be analyzed in parallel, i.e., the detection rate. Recall that SoftATS is multi-threaded (Sec. IV). **Setup:** The evaluation was carried out on a 64 bit Intel Core i7-3517U CPU @ 1.90 GHz with 4GB of RAM. We are interested in measuring the throughput of computing trajectories, therefore, we use multiple detector threads. Hence, in addition to the default parameters as mentioned in Sec. VII-A, we use the following: **Detectors** t: 2, 4, 6 and 8. Note that each detector thread is an independent thread within SoftATS, that can compute trajectories and detect attacks.

Methodology: To measure the detection throughput (trajectories/s) of the multi-threaded SoftATS, we record the total time taken for a single sample to be dispatched to its respective detector and for the detector to complete the trajectory computation. Each detector thread computes 1k trajectories, from which we compute the mean throughput for t detector threads, which is given by Eq. 2. We then repeat the measurements for different CPU core counts (1, 2 and 4/hyper-Threading).

$$throughput = \frac{t}{mean(DispatchTime + DetectionTime)} \quad (2)$$

Results and Analysis: The results obtained are shown in Fig. 8. It is clearly evident that there is an increase in the throughput with an increase in detector threads. However, the throughput does not increase linearly with the thread count and the number of cores. Nonetheless, there is a performance improvement. Using 8 threads and a hyper-threaded CPU, the throughput is close to 3000 trajectories/s which is ∼3 times that of using a single thread on a single core CPU. The non-linear increase can be attributed to the History list (see Detection Algorithm 3), where read and write operations are synchronized. Furthermore, we suspect that: (i) ONOS itself requires CPU cycles to manage and maintain itself and the switches, (ii) I/O interrupts cause the operating system to preempt SoftATS and ONOS, and (iii) the CPU architecture used for this experiment uses hyperthreading (Intel's implementation of simultaneous multithreading), influencing the results non-deterministically. Nevertheless, the results lend credence to the use of multiple detection threads to achieve high detection rates and high availability. Furthermore, the use of multiple threads on multiple collectors can substantially increase the detection throughput.
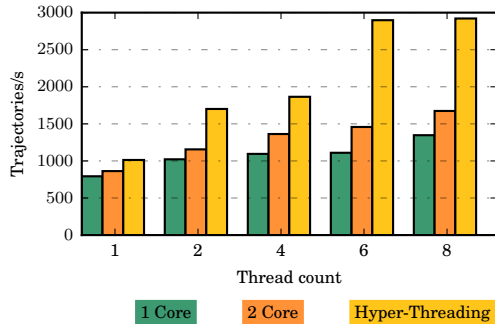
Fig. 8. Comparison of SoftATS mean throughput (trajectories/s) with varying thread counts and CPU cores.

Fig. 9. Resource consumption of SoftATS.

D. Overhead of Sampling

Although SoftATS has benefits in adversarial environments, there are potential overheads. In the following we measure and analyze the overhead of sampling at the (ONOS) controller/collector, and the switch (OvS). We first characterize the overhead in terms of CPU and memory usage at the controller. Next, we measure the overhead, if any, sampling introduces to the forwarding throughput and latency of different sized network packets.

1) Overhead at the Controller: For overhead at the controller, we are primarily interested in measuring the CPU and memory utilization of SoftATS. We begin by describing our experimental setup, followed by our methodology, and then conclude with the results and analysis.

Setup: We use the same system from the detection throughput experiment for characterizing the overhead of sampling at the controller. For SoftATS, we use the default parameters as mentioned in Sec. VII-A.

Methodology: To characterize the CPU and memory consumption at the collector, we first measure the resource consumption without SoftATS, then enable a static configuration of SoftATS, and finally we enable the dynamic configuration of SoftATS. In all the cases, we replay bi-directional traffic at 100 pps between 6 pairs of hosts, i.e., each pod receives 3 flows, for 10 minutes. We measure the CPU and memory consumption using *top* every second with the controller pinned to a single core.

Results and Analysis: Fig. 9 shows the results. Naturally, SoftATS does increase CPU and memory consumption on average compared to the baseline usage. The CPU increase is due to SoftATS dispatching a sample to the detector thread(s) and computing the trajectory. The memory increase by approximately 1.5 MB is due to the data structures used by SoftATS to store and process samples and hash assignments for switches. 50% of the CPU usage for the static and dynamic cases is within 12-35% usage, and 50% of the memory usage is within 14-15.5 MB: both are within acceptable boundaries for
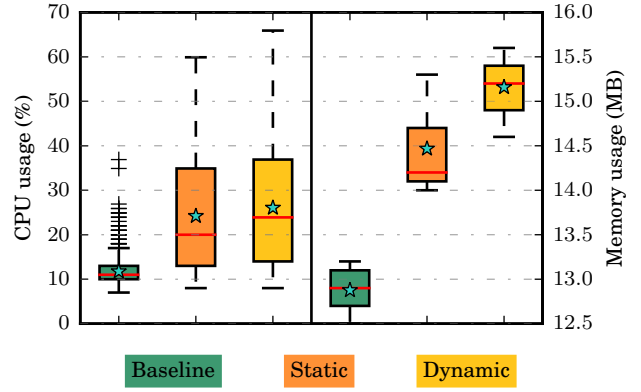
scenarios where SoftATS executes on a controller that manages the network. Furthermore, the dynamic configuration, even with an average update rate of 2s, does not increase the mean CPU and memory usage by much, compared to the static configuration. This implies that the benefits of the dynamic configuration can be reaped with minor overheads in terms of CPU and memory.

If SoftATS were to be used as a stand-alone system or with multiple collectors, these overheads are very low. In fact, as we have demonstrated previously in Sec. VII-C, the detection throughput can be increased by using several detector threads, and the load can be distributed across several collectors, thereby reducing the resource consumption on individual collectors. Hence, we see the overhead at the controller to be a small trade-off for the benefits of a highly parallel, distributed and performant detection system such as SoftATS.

2) Overhead at the Switch: Having investigated the overhead at the controller, we now measure the overhead of sampling at the switch. The two main metrics we are interested in here are: throughput, and latency. Using a traffic generator and hardware timestamps, we are able to accurately and precisely measure the impact of sampling on the throughput and latency of SoftATS at the switch. In the following, we describe our experimental setup for evaluating the impact of sampling on the forwarding performance. Due to our limited resources, we evaluate the overhead using a software switch, namely, Open vSwitch.

Setup: Since we are interested in the forwarding performance of the switch when SoftATS is used, we use one server running OvS. Therefore, the topology in this section uses only one switch and one controller. In addition to the default parameters from Sec. VII-A, we use the following: **Sampling Ratio** p_s**:** 0%, 0.9% and 1.3%. We only use the static assignment scheme in this experiment. To measure the throughput and latency of OvS, we use five systems as shown in Fig. 10. We use a Traffic Generator—to generate specific sized packets,
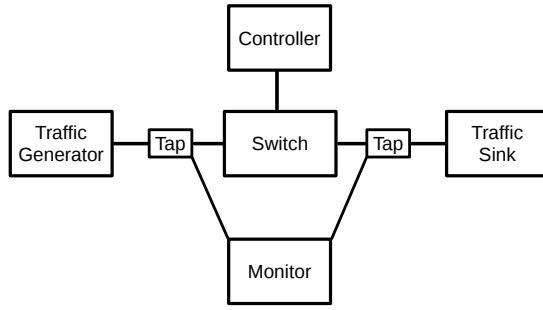
Fig. 10. Topology used to measure the forwarding performance of OvS.

Fig. 11. Forwarding throughput of 64 byte packets for different sampling ratios.

and replay them at specific rates to OvS. OvS forwards packets to the Traffic Sink. OvS is also connected to the Controller via a dedicated interface. Using passive network taps on the ingress and egress links of OvS, we collect the traffic at the Monitor system. Except for the Monitor, all other systems use two 2.5 GHz Dual Core AMD Opteron CPUs with 16 GB of RAM, and four Intel Gigabit Network Interface Cards (NICs). The Monitor uses two dual-core 3.7 GHz Intel Xeon CPUs with hyper-threading enabled and 16 GB of RAM. We use an Endace DAG 10X4-P card to capture network packets on the Monitor. Each interface of the card has a receive queue configured with 1 GB. This provides us with accurate, and highly precise uni-directional measurements, independent of the host system's (Monitor) resource utilization. All the systems run Ubuntu 14.04. For OvS, we use ovs-2.3.2 compiled using the default configure script with gcc. OvS also has a kernel-based fast-path, we use version: 20D84E92C1F09E01E1586EE running on a Linux kernel version 4.6.5.

Furthermore, we configure OvS with one rule for sampling, and one rule for forwarding, apart from the two default flow rules installed by ONOS's proxyarp, and reactive forwarding application. The sampling rule matches VLAN tags only, while the forwarding rule matches packets based on the Ethernet source and destination MAC addresses. The transmitted packets, are UDP packets encapsulated in IP packets. Ethernet is used to encapsulate the IP packets.

Methodology: Our objective is to measure the uni-directional forwarding performance overhead, if any, of OvS using our sampling scheme. Therefore, we first measure the forwarding performance of OvS without our scheme, i.e., **Sampling Ratio** p_s**:** 0, and then with sampling, i.e., p_s**:** 0.4%, 0.9% and 1.3%.

To measure the throughput, we transmit a constant stream of UDP packets at a specific rate and measure the rate at which the stream is forwarded by OvS at the Monitor. To unearth performance bottlenecks quickly,
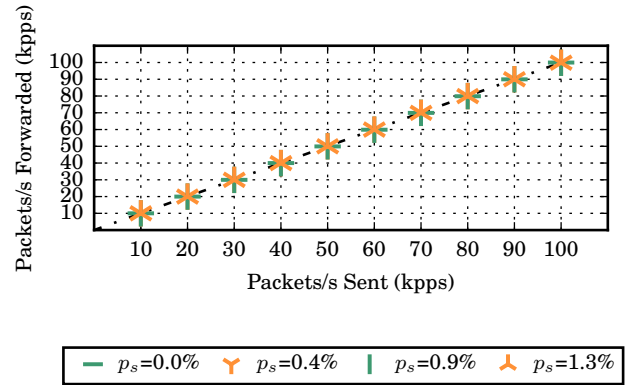
we use packets per second (pps) as the metric for throughput [49]. To identify the impact of packet sizes on the throughput, we use three packet (frame) sizes (Bytes) based on the IEEE 802.3 Ethernet standard for basic frames [50]: 64, 512 and 1518. For each packet size, we transmit packets for 330 seconds at a constant rate. We then reset the setup and change the transmit rate. The transmit rate starts at 10 kpps and goes up to 100 kpps, in steps of 10 k.

To measure the latency, we transmit specific sized packets and measure the time taken for OvS to forward them. We do so for packet sizes (Bytes) based on the Ethernet standard: 64, 128, 256, 512, 1024, 1280 and 1518. For each trial, we send 10500 specific sized packet at 100 pps. We discard the first 500 packets to eliminate any system artefacts.

Results and Analysis: Fig. 11, 12, and 13 show the overhead of sampling for different sized packets. For 64 and 512 byte packets, OvS could reach 100 kpps transmission rates, however, for 1518 byte packets, the peak transmission rate was only 70 kpps. The results show that for the sampling ratios chosen, the forwarding throughput of OvS is not impacted.

Fig. 14 shows the forwarding latency of OvS for different sized packets without and with sampling. It is evident that sampling introduces many outliers. This is because sampling increases the packet processing pipeline compared to plain forwarding. When a packet matches the sampling rule in OvS it has to go through OvS's slow path. Therefore, it has to go from the kernel-space to user-space for processing. The action of sending the packet to the controller increases the packet processing. Lastly, the packet is matched against two tables until it is forwarded, as opposed to only one table in the baseline. From the figures we can also see that regardless of sampling, for 1024 byte packets and higher, the latency increase by roughly 25%.
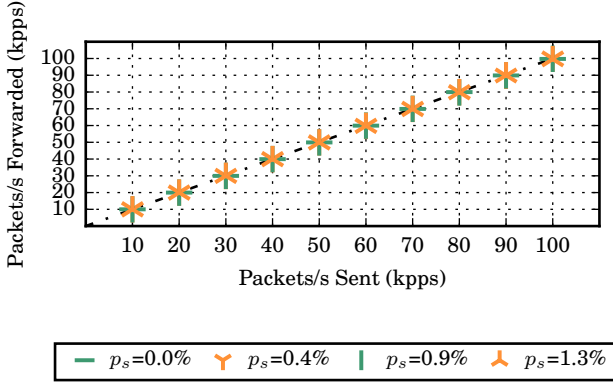
Fig. 12. Forwarding throughput of 512 byte packets for different sampling ratios.
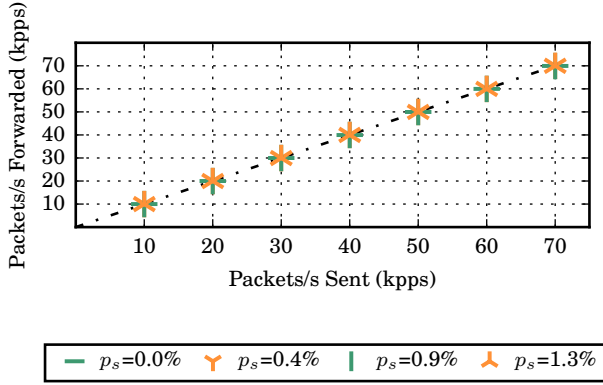

Fig. 13. Forwarding throughput of 1518 byte packets for different sampling ratios.
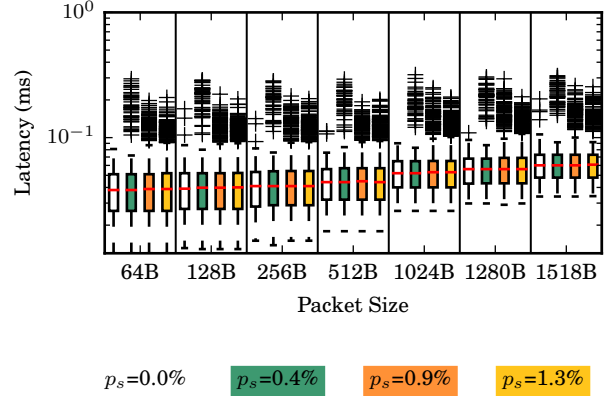

Fig. 14. Forwarding latency of different sampling ratios using various sized packets at 100 pps.
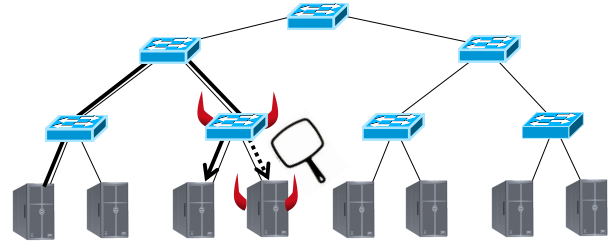
Fig. 15. Detecting attacks by the top-of-the-rack switch is hard.

VIII. DISCUSSION

Our approach raises opportunities for further improvements, but also comes with limitations. We will discuss some of them in more detail in the following.

A. Sampling Extension to the Edge

Our adversarial trajectory sampling scheme comes with the fundamental limitation that the network edge must be trusted: if a packet does not even enter the sampling network, it is impossible to detect a routing error related to it. For example, consider the scenario depicted in Figure 15: As there are no sample points on the route after the malicious ToR switch, it is impossible to detect the mirrored packet to the malicious host.

However, the problem can be alleviated if the sampling scheme is extended to the edge. For example, in cloud management systems such as OpenStack, Open vSwitch (OvS) is used on the virtualized servers which is already OpenFlow enabled. Assuming that the virtualized server's host OS can be trusted, OvS (or the networking stack on the host OS) can also participate in our scheme to detect misbehaving ToR switches.

Fig. 16 compares the packets till detection when the servers are used, and not used in SoftATS. We indicate detecting the aggregate switch as the worst case, and detecting the core switch as the best case. This is because the aggregate switch has the least number of pairs surrounding it, and the core switch has the most number of pairs surrounding it. Clearly, there is a benefit in using the servers with respect to detecting attacks. The servers improve the detection time by a factor of roughly 2x. By adding the servers to SoftATS, the total number of switches increase from 20 to 36 which increases the explicit pairs across the network. Furthermore, including the servers increases the number of switches surrounding the attacker by two, thereby increasing the detection probability.

One interesting challenge of this approach regards how to securely connect OvS with the collector: this connection needs to be implemented in-band, via other (possibly) malicious (ToR) switches.

An alternative approach to extend the security to the edge of the network could be to use a robust combiner approach [51], [35], i.e., by leveraging physical redundancy (e.g., switches of different vendors or switches which were fabricated in different countries).
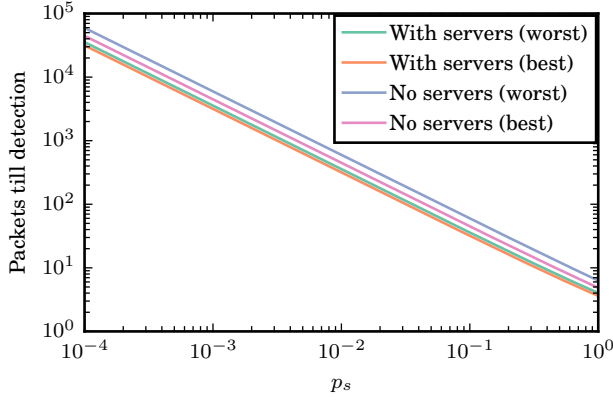
Fig. 16. A comparison of the number of packets to detection with and without including the edge servers in SoftATS in the best and worst case.
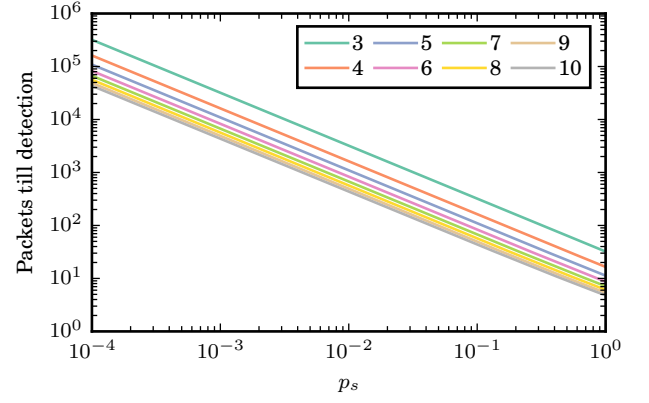

Fig. 17. A comparison of the number of packets to detection for the worst case in ATT's network for various path lengths.

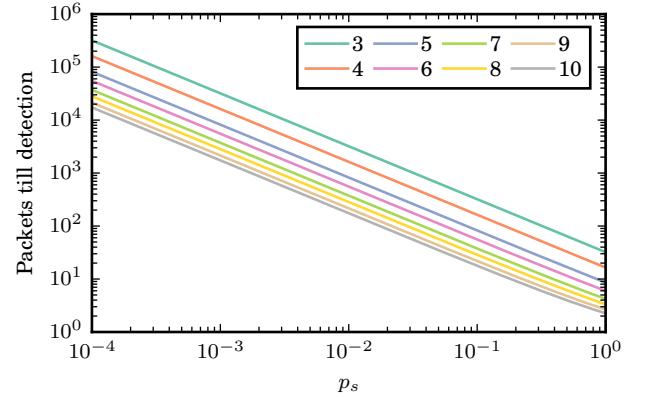Fig. 18. A comparison of the number of packets to detection for the best case in ATT's network for various path lengths.

B. Applying SoftATS to Other Networks

So far we have used a data center network as our primary topology of SoftATS. We observe that SoftATS is by no means limited to such networks. It can also be applied to wide area networks (WAN) and service provider networks. Admittedly, to use SoftATS in the Internet, all nodes across end-points need to participate in our scheme. Accomplishing this, is no small feat and is beyond the scope of this paper. However, it is still feasible for SoftATS to be used within an administrative domain such as an internet service provider for example.

We expect no modifications to SoftATS when used in a service provider network with the exceptions of obtaining centralized control, and global routing information. In fact, we observe that using SoftATS, one can obtain fine-grained monitoring over the network, as the sampling can be applied at switches (L2) and routers (L3) in unison. In addition, we expect WANs and service provider networks to have long paths. Therefore, in the following we analyse the relationship between the detection probabilities and the path length using ATT's network topology.

The ATT network we use [52], comprises of a total of 54 nodes with a median path length of 7 [15]. Such a network gives us room to explore how the detection probability and path lengths interact. This was not possible in the Clos topology we used for our evaluation. Fig. 17 and 18, show the relationship of the detection probability to path lengths for the worst and best case resp. The worst case for a given path is the attacker's position that has the fewest pairs surrounding it, i.e., the node adjacent to the edge node. The best case is the attacker's position that has the most pairs surrounding it, i.e., the middle node on the path. From both figures it is clear that the packets till detection and the sampling ratio grows linearly. Furthermore, it appears that increasing the path length reduces the packets till detection exponentially only till a certain length beyond

which there is little benefit in long paths. Nonetheless, it shows that longer paths and more nodes, do indeed increase the detection probability.

C. Localization Is Difficult

While our sampling scheme is useful to *detect* misbehavior, it is hard to *localize* a misbehaving switch. To illustrate this, let us consider some examples. When two switches that are far apart (e.g., the source and destination switches on a long path), share an assignment that generates an alert, it is difficult to isolate the malicious switch: it could be any of the switches after the source and before the destination switch. Localization is particularly hard for injection attacks: it is difficult to determine the misbehaving switch, if the packet is not sampled at all the switches.

Nevertheless, each detected misbehavior gives some clues where things went wrong. Accordingly, due to our time-varying sampling strategy, there is always a chance that leveraging tomographic techniques to perform localization is actually possible for a concrete event. Over time, such patterns may become statistically significant.

D. Attacks on the Collector

SoftATS is built upon the assumption that it is hosted on a secure system. Therefore, if an attacker compromises the system that hosts SoftATS then our scheme can be broken. Hence, appropriate measures (which are beyond the scope of this paper) must be taken to ensure that the system hosting SoftATS is sufficiently secured.

The availability of our scheme relies on the system(s) running the collector(s), the link capacities between the switches and the collector(s), etc. Indeed an attacker can overwhelm the collector with samples and then carry out the attack. However, there already exist measures to deal with such problems, e.g., using redundant and multiple collectors. By rate-limiting the samples arriving from the switch, the collector may also be protected from a dos attack. To ensure availability of the scheme and the network, it is best for SoftATS to run on dedicated and multiple controllers.

The attacker can reorder transit packet thereby influence the order of the samples reported to the collector. If the delay in receiving the sample is beyond the *max_delay*, then SoftATS will throw two alerts. One drop attack alert for the first sample and one inject attack alert for the second sample. It will not miss detecting such an attack.

IX. RELATED WORK

Being able to measure routes and traffic is critical for controlling and engineering a communication network [53]. Accordingly, the topic has been studied intensively in the past. One generally distinguishes between indirect and direct measurement methods. An indirect measurement method relies on a network model and network status information to infer the traffic route (e.g., IGP and BGP routing state). Indirect measurement methods suffer from the uncertainty associated with the physical and logical state of the network. In contrast, direct methods rely on direct observations of traffic at multiple points in the network. Trajectory sampling [26], [27] is an example of the latter.

Many operating systems support the *traceroute* and *ping* [54], [55] command to display the route (path) and to measure transit delays of packets across an Internet Protocol (IP) network. These tools are based on active test packet injection and are attractive for their parsimonious use of data plane resources. However, it is well-known that ping, traceroute, and counter-based solutions [56] fail in adversarial settings [53]: For example, a malicious switch may fake responses to provide the impression that probing packets traverse the network properly. Both active [54], [55] and passive probing [26], [27] are vulnerable to forged reports. In general, any adversarial behavior that can maliciously misreport samples, or delete or modify information contained in packet headers, essentially breaks a scheme based on packet labeling or tagging [30], [31].

Trajectory Sampling [26], [27] is a direct measurement method based on a consistent sampling approach allowing to reconstruct the path, at the collector, given a pseudorandom subset of packets through the corresponding network domain. The approach is attractive as most switch vendors today implement some form of packet sampling (e.g., Netflow [57]). However, while trajectory sampling may still perform well in the context of faulty networks [28] and networks with different flow sizes [29], it is insufficient in non-cooperative and adversarial environments.

Interestingly, while over the last years, much research was conducted on how to secure routing protocols on the control plane [1], [2], [3], [4], [58], providing authenticity and correctness of topology propagation and route computation, the important question of how to secure the data plane has received much less attention so far. In fact, until very recently, researchers did not know whether it is possible to build a secure path verification mechanism [22]. Many existing systems like VeriFlow [59], Anteater [60] and Header Space Analysis [23] rely either on flow rules installed at switches or on data plane configuration information to perform their analysis. This information can easily be manipulated in malicious settings. Providing fundamental properties like path consent and path compliance [22] are usually based on cryptographic techniques: expensive operations in high speed networks. Even more challenging than inferring the routes along which certain packets actually travelled is to test where packets *did not travel* [61].

Goldberg et al. [53] propose an interesting approach for end-to-end path quality monitoring; while this approach is suitable also in very general settings, it comes at the cost of introducing extra information in packets and requiring stronger hash functions. In an SDN setting where communication between switches and controllers are out-of-band and secured, opportunities for simpler solutions such as ours are introduced.

Our approach is particularly motivated by the introduction of software-defined networks. While SDNs are known to introduce many flexibilities, also in terms of security, they also introduce new threats. For instance, Yu et al. [62] presented a distributed traffic monitoring scheme for SDNs, and FleXam [63] is a sampling extension for monitoring and security applications in OpenFlow. NetSight [64] leverages SDN to trace entire packet histories (without sampling), by collecting them "out-of-band", and CherryPick [65] uses packets to carry information of SDN paths "in-band" (namely, a subset of links along the packet trajectory); however, these protocols struggle with drops and are not robust to malicious switches. In particular, the information CherryPick adds to the header along the path is only verified at the end of the path. Bates et al. [66] use SDN networks (plus some middleboxes) to observe the data plane behavior, even in the presence of malicious switches. The traffic

engineering flexibilities of SDN have also been exploited to perform secret sharing [67]. Zeng et al. [55] use SDN to test the forwarding and policies in the network by generating and actively probing the data plane across the network.

Desai et al. [68] propose a hash-based delay sampling technique to detect switches misforwarding packets. Their threat model and objectives are closely related to ours. However, their sampling scheme and detection algorithm are different from ours. They require three switches to sample the same set of packets and their detection algorithm depends on the state of the switches buffers for a chosen path. We can incorporate their method of sampling i.e., choose triplets instead of pairs, but our detection is based on trajectories and not switch states.

Yu et al. [69] describe OpenSketch, a generic and efficient sketch-based measurement framework for SDN data planes. They developed APIs and sketches to make generic SDN measurements alleviate the control plane programming complexity operators face. Additionally, they present a prototype using NetFPGA to demonstrate the feasibility, applicability and overhead of their approach. Since their framework is designed to be parallel to the packet processing pipeline, there is no performance impact to the forwarding. However, matching OpenFlow flow rules and sending samples to the controller are not feasible. Nonetheless, their framework can be modified to implement our scheme.

Bu et al. [70] introduce accurate and efficient algorithms to detect and troubleshoot flow rule and flow table faults in the data plane. They do so by using probe packets through the data plane. This approach is vastly different from ours. We rely on a passive approach, therefore we do not send probe packets through the data plane. Moreover, our objective is to detect incorrect packet trajectories rather than incorrect forwarding rules and tables that may cause incorrect trajectories. Furthermore, our threat model does not restrict the cause for misforwarding to only flow rules and flow priorities even though it can be used to do so.

The paper most closely related to ours is by Lee et al. [15]. The authors present a smart generalization of trajectory sampling where hash values are shared between a subset of switches, rendering it hard for an adversary to avoid detection. The authors also present a first simulation of the possibility to detect packet drop, modifications and substitution attacks. Our paper builds upon this work in multiple respects. We initiate the study of injection attacks and observe that if combined with drop attacks, injection attacks introduce a number of more sophisticated attacks. Accordingly, we extend and formally and empirically analyze detection algorithm guarantees under various misbehaviors, including injections, mirroring, rerouting, or modifications of headers and/or payloads. Unlike prior work, our hash assignment algorithm is completely random, eliminating bias for pairs and their assigned hash values. Moreover, our detection algorithm does not rely on an aggregation of trajectories and counters. Instead, we use the collector's (controller's) global view of the network (i.e., the network policy oracle) to compute every sampled packet's trajectory which gives us a per sample detection accuracy rather than an aggregate. We demonstrate that our detection algorithm can be parallelized using multiple threads (and possibly multiple collectors).

In general, our work is motivated by the insight that SDNs provide an ideal environment to implement our algorithms, and we present an OpenFlow prototype accordingly. We understand SoftATS as a flexible framework which supports a wide range of algorithms, tailored towards their specific settings and threat models.

X. CONCLUSION

Today's computer network routing protocols incorrectly assume that switches are non-malicious, and accordingly, a sufficiently large portion of the computer network infrastructure is vulnerable to security attacks from compromised switches, e.g., containing hardware or software backdoors [58].

We in this paper presented a simple and light-weight adversarial sampling approach based on the software-defined networking paradigm, which can detect a wide range of adversarial behaviors, and in different settings, e.g., in the datacenter but also in the wide-area network [71]. Our scheme cannot only detect packet drops, but also packet injections, rerouting, or even packet headers and content alterations.

Furthermore, we implemented our SoftATS in the standard OpenFlow protocol, evaluated the prototype in terms of detection time, and detection throughput. We also measured the overhead introduced by SoftATS and identified that at the controller there is a modest increase in CPU and memory utilization. At the switch, there is little to no forwarding overhead, making SoftATS feasible in the real world.

We believe that our work opens several interesting directions for future work. In particular, our framework allows to experiment with many alternative algorithms, which can be optimized and tailored towards specific use cases (e.g., enterprise networks) and attacks. Moreover, we have so far focused on detection only, and many interesting questions for future research arise when aiming at the *localization* and *prevention* of malicious nodes and/or behaviors.

REFERENCES

[1] W. Aiello, J. Ioannidis, and P. McDaniel, "Origin authentication in interdomain routing," in *Proc. ACM Conference on Computer and Communications Security (CCS)*, 2003, pp. 165–178.

[2] K. Butler, T. Farley, P. McDaniel, and J. Rexford, "A Survey of BGP Security Issues and Solutions," *Proceedings of the IEEE*, vol. 98, no. 1, pp. 100–122, 2010.

[3] Y.-C. Hu and A. Perrig, "A Survey of Secure Wireless Ad Hoc Routing," *Proc. IEEE Security & Privacy (S&P)*, vol. 2, no. 3, pp. 28–39, May 2004.

[4] S. Kent, C. Lynn, and K. Seo, "Secure Border Gateway Protocol (S-BGP)," *IEEE J. Selected Areas in Communications*, vol. 18, no. 4, pp. 582–592, September 2006.

[5] M. Antikainen, T. Aura, and M. Särelä, "Spook in Your Network: Attacking an SDN with a Compromised OpenFlow Switch," in *Secure IT Systems*. Springer, 2014, pp. 229–244.

[6] P.-W. Chi, C.-T. Kuo, J.-W. Guo, and C.-L. Lei, "How to detect a compromised sdn switch," in *Proc. IEEE NetSoft*, 2015.

[7] M. Dhawan, R. Poddar, K. Mahajan, and V. Mann, "Sphinx: Detecting security attacks in software-defined networks." in *Proc. Internet Society Symposium on Network and Distributed System Security (NDSS)*, 2015.

[8] A. Kamisiński and C. Fung, "Flowmon: Detecting malicious switches in software-defined networks," in *Proc. ACM Workshop on Automated Decision making for Active Cyber Defense*, 2015, pp. 39–45.

[9] K. Thimmaraju, L. Schiff, and S. Schmid, "Outsmarting network security with sdn teleportation," in *Proc. IEEE European Security & Privacy (S&P)*, 2017.

[10] "A simple command allows the CIA to commandeer 318 models of Cisco switches," https://arstechnica.com/security/2017/03/a-simple-command-allows-the-cia-to-commandeer-318-models-of-cisco-switches/, 2017, accessed: 21-03-2017.

[11] "The Tale of One Thousand and One DSL Modems," https://securelist.com/analysis/publications/57776/the-tale-of-one-thousand-and-one-dsl-modems/, 2012, accessed: 9-03-2017.

[12] "SYNful Knock - A Cisco router implant - Part I," https://www.fireeye.com/blog/threat-research/2015/09/synful_knock_-_acis.html, 2015, accessed: 9-03-2017.

[13] F. Lindner, "Cisco ios router exploitation," *Black Hat USA*, 2009.

[14] D. Chasaki and T. Wolf, "Attacks and defenses in the data plane of networks," in *Proc. IEEE/IFIP Transactions on Dependable and Secure Computing (DSN)*, vol. 9, no. 6, Nov 2012, pp. 798–810.

[15] S. Lee, T. Wong, and H. S. Kim, "Secure split assignment trajectory sampling: A malicious router detection system," in *Proc. IEEE/IFIP Transactions on Dependable and Secure Computing (DSN)*, 2006, pp. 333–342.

[16] "Huawei HG8245 backdoor and remote access," http://websec.ca/advisories/view/Huawei-web-backdoor-and-remote-access, 2013, accessed: 9-03-2017.

[17] "Netis Routers Leave Wide Open Backdoor," http://blog.trendmicro.com/trendlabs-security-intelligence/netis-routers-leave-wide-open-backdoor/, 2014, accessed: 9-03-2017.

[18] "Snowden: The NSA planted backdoors in Cisco products," http://www.infoworld.com/article/2608141/internet-privacy/snowden--the-nsa-planted\-backdoors-in-cisco-products.html, 2014, accessed: 9-03-2017.

[19] K. Thimmaraju, B. Shastry, T. Fiebig, F. Hetzelt, J.-P. Seifert, A. Feldmann, and S. Schmid, "Reigns to the cloud: Compromising cloud systems via the data plane," *arXiv preprint arXiv:1610.08717*, 2016.

[20] G. Mathur, V. N. Padmanabhan, and D. R. Simon, "Securing Routing in Open Networks Using Secure Traceroute," Microsoft Research, Tech. Rep. MSR-TR-2004-66, July 2004.

[21] Federal Offic for Information Security (BSI), "The State of IT Security in Germany 2015," https://www.bsi.bund.de/SharedDocs/Downloads/EN/BSI/Publications/Securitysituation/IT-Security-Situation-in-Germany-2015.pdf, BSI, Tech. Rep., 2015, accessed: 9-03-2017.

[22] J. Naous, M. Walfish, A. Nicolosi, D. Mazières, M. Miller, and A. Seehra, "Verifying and Enforcing Network Paths with Icing," in *Proc. ACM CoNEXT*, 2011, pp. 30:1–30:12.

[23] P. Kazemian, G. Varghese, and N. McKeown, "Header space analysis: Static checking for networks," in *Proc. Usenix Symposium on Networked Systems Design and Implementation (NSDI)*, 2012, pp. 113–126.

[24] HIPAA, "The Health Insurance Portability and Accountability Act," https://www.hhs.gov/hipaa/for-professionals/security/laws-regulations/index.html, 2014, accessed: 9-03-2017.

[25] K. Zetter, "Google Hack Attack Was Ultra Sophisticated, New Details Show," https://www.wired.com/2010/01/operation-aurora/, 2010, accessed: 9-03-2017.

[26] N. G. Duffield and M. Grossglauser, "Trajectory sampling for direct traffic observation," *IEEE/ACM Trans. Networking (TON)*, vol. 9, no. 3, pp. 280–292, June 2001.

[27] N. Duffield, A. Gerber, and M. Grossglauser, "Trajectory engine: a backend for trajectory sampling," in *Proc. IEEE/IFIP Symposium on Network Operations and Management (NOMS)*, 2002, pp. 437–450.

[28] N. Duffield and M. Grossglauser, "Trajectory sampling with unreliable reporting," *Networking, IEEE/ACM Transactions on*, vol. 16, no. 1, pp. 37–50, 2008.

[29] V. Sekar, M. K. Reiter, W. Willinger, H. Zhang, R. R. Kompella, and D. G. Andersen, "CSAMP: A System for Network-wide Flow Monitoring," in *Proc. Usenix Symposium on Networked Systems Design and Implementation (NSDI)*, 2008, pp. 233–246.

[30] S. Narayana, J. Rexford, and D. Walker, "Compiling Path Queries in Software-defined Networks," in *Proc. ACM Workshop on Hot Topics in Software Defined Networking (HotSDN)*, 2014, pp. 181–186.

[31] Y. Zhang, M. Roughan, N. Duffield, and A. Greenberg, "Fast Accurate Computation of Large-scale IP Traffic Matrices from Link Loads," in *Proc. ACM SIGMETRICS*, 2003, pp. 206–217.

[32] A. Feldmann, A. Greenberg, C. Lund, N. Reingold, and J. Rexford, "NetScope: Traffic engineering for IP networks," *IEEE Network Magazine*, vol. 14, no. 2, pp. 11–19, 2000.

[33] M. Al-Fares, A. Loukissas, and A. Vahdat, "A Scalable, Commodity Data Center Network Architecture," in *Proc. ACM SIGCOMM*, 2008, pp. 63–74.

[34] D. Achenbach, J. Müller-Quade, and J. Rill, "Universally composable firewall architectures using trusted hardware," in *Proc. International Conference on Cryptography and Information Security in the Balkans*, 2014, pp. 57–74.

[35] A. Feldmann *et al.*, "Netco: Reliable routing with unreliable routers," in *IEEE Workshop on Dependability Issues on SDN and NFV*, 2016.

[36] T. Mizrahi and Y. Moses, "Time4: Time for sdn," *IEEE Transactions on Network and Service Management*, vol. 13, no. 3, pp. 433–446, Sept 2016.

[37] Y. Afek, A. Bremler-Barr, and L. Schiff, "Ranges and cross-entrance consistency with openflow," in *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking*, ser. Proc. ACM Workshop on Hot Topics in Software Defined Networking (HotSDN). ACM, 2014, pp. 233–234.

[38] "Open vSwitch Manual," http://openvswitch.org/support/dist-docs/ovs-ofctl.8.txt, 2017, accessed: 9-03-2017.

[39] Y. Afek, A. Bremler-Barr, S. Landau Feibish, and L. Schiff, "Sampling and Large Flow Detection in SDN," in *Proc. ACM SIGCOMM Poster*, 2015, pp. 345–346.

[40] M. Kuźniar, P. Perešíni, and D. Kostić, "What you need to know about SDN flow tables," in *Proc. Passive and Active Measurement (PAM)*. Springer, 2015, pp. 347–359.

[41] M. C. Wendl, "Collision probability between sets of random variables," *Statistics & Probability Letters*, vol. 64, no. 3, pp. 249 – 254, 2003. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167715203001688

[42] W. E. Yancey, "Expected Number of Random Duplications Within or Between Lists," in *Proc. Section on Survey Research Methods, American Statistical Association*, 2010.

[43] (2016) The Poisson Distribution. http://www.math.csusb.edu/faculty/stanton/probstat/poisson/Distribution.java. Accessed: 9-03-2017.

[44] M. Shahbaz, S. Choi, B. Pfaff, C. Kim, N. Feamster, N. McKeown, and J. Rexford, "PISCES: A Programmable, Protocol-Independent Software Switch," in *Proc. ACM SIGCOMM*. ACM, 2016, pp. 525–538.

[45] B. Heller. (2013) RipL: a Python library to simplify the creation of data center code. https://github.com/brandonheller/ripl. Accessed: 9-03-2017.

[46] M. Casado *et al.*, "Ripcord: A Modular Platform for Data Center Networking," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2010-93, Jun 2010. [Online]. Available: http://www.eecs.berkeley.edu/Pubs/TechRpts/2010/EECS-2010-93.html

[47] R. Pang, M. Allman, M. Bennett, J. Lee, V. Paxson, and B. Tierney, "A first look at modern enterprise traffic," in *Proc. ACM Internet Measurement Conference*, 2005, pp. 2–2.

[48] C. Partridge, J. Hughes, and J. Stone, "Performance of checksums and crcs over real data," in *ACM Computer Communication Review (CCR)*, vol. 25, no. 4. ACM, October 1995, pp. 68–76.

[49] V. Jacobson, "Congestion avoidance and control," in *ACM Computer Communication Review (CCR)*, vol. 18, no. 4, 1988, pp. 314–329.

[50] IEEE, "IEEE Standard for Ethernet," IEEE Standards Association, IEEE, Std, 2015. [Online]. Available: http://www.ieee802.org/3/

[51] D. Harnik, J. Kilian, M. Naor, O. Reingold, and A. Rosen, "On Robust Combiners for Oblivious Transfer and Other Primitives," in *Advances in Cryptology EUROCRYPT*, ser. Lecture Notes in Computer Science, R. Cramer, Ed., 2005, vol. 3494.

[52] "Tier-1 Internet provider networks with multiple peering points," http://www.ssfnet.org/Exchange/gallery/usa/index.html, 2002, accessed: 30-04-2017.

[53] S. Goldberg, D. Xiao, E. Tromer, B. Barak, and J. Rexford, "Path-quality Monitoring in the Presence of Adversaries," in *Proc. ACM SIGMETRICS*, 2008, pp. 193–204.

[54] K. Agarwal, E. Rozner, C. Dixon, and J. Carter, "SDN Traceroute: Tracing SDN Forwarding Without Changing Network Behavior," in *Proc. ACM Workshop on Hot Topics in Software Defined Networking (HotSDN)*, 2014, pp. 145–150.

[55] H. Zeng, P. Kazemian, G. Varghese, and N. McKeown, "Automatic test packet generation," in *Proc. ACM CoNEXT*, 2012, pp. 241–252.

[56] L. Subramanian, V. Roth, I. Stoica, S. Shenker, and R. H. Katz, "Listen and Whisper: Security Mechanisms for BGP," in *Proc. Usenix Symposium on Networked Systems Design and Implementation (NSDI)*, 2004.

[57] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin, *Network Flows: Theory, Algorithms, and Applications*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1993.

[58] L. Subramanian, "Decentralized security mechanisms for routing protocols," Ph.D. dissertation, University of California at Berkeley, 2005.

[59] A. Khurshid, X. Zou, W. Zhou, M. Caesar, and P. B. Godfrey, "VeriFlow: Verifying Network-Wide Invariants in Real Time," in *Proc. Usenix Symposium on Networked Systems Design and Implementation (NSDI)*, 2013.

[60] H. Mai, A. Khurshid, R. Agarwal, M. Caesar, P. B. Godfrey, and S. T. King, "Debugging the data plane with anteater," in *Proc. ACM SIGCOMM*, 2011, pp. 290–301.

[61] D. Levin, Y. Lee, L. Valenta, Z. Li, V. Lai, C. Lumezanu, N. Spring, and B. Bhattacharjee, "Alibi Routing," in *Proc. ACM SIGCOMM*, 2015, pp. 611–624.

[62] Y. Yu, C. Qian, and X. Li, "Distributed and Collaborative Traffic Monitoring in Software Defined Networks," in *Proc. ACM Workshop on Hot Topics in Software Defined Networking (HotSDN)*, 2014, pp. 85–90.

[63] S. Shirali-Shahreza and Y. Ganjali, "FleXam: Flexible Sampling Extension for Monitoring and Security Applications in Openflow," in *Proc. ACM Workshop on Hot Topics in Software Defined Networking (HotSDN)*, 2013, pp. 167–168.

[64] N. Handigol, B. Heller, V. Jeyakumar, D. Mazières, and N. McKeown, "I Know What Your Packet Did Last Hop: Using Packet Histories to Troubleshoot Networks," in *Proc. Usenix Symposium on Networked Systems Design and Implementation (NSDI)*, 2014, pp. 71–85.

[65] P. Tammana, R. Agarwal, and M. Lee, "CherryPick: Tracing Packet Trajectory in Software-defined Datacenter Networks," in *Proc. ACM Symposium on Software Defined Networking Research (SOSR)*, 2015.

[66] A. Bates, K. Butler, A. Haeberlen, M. Sherr, and W. Zhou, "Let SDN Be Your Eyes: Secure Forensics in Data Center Networks," in *Proc. NDSS Workshop on Security of Emerging Network Technologies (SENT'14)*, February 2014.

[67] S. Dolev and S. Tzur-David, "SDN-Based Private Interconnection," in *Proc. ACM PODC Workshop on Distributed Software-Defined Networks (DSDN)*, 2014.

[68] V. Desai, S. Natarajan, and T. Wolf, "Packet forwarding misbehavior detection in next-generation networks," in *2012 IEEE International Conference on Communications (ICC)*. IEEE, 2012, pp. 846–851.

[69] M. Yu, L. Jose, and R. Miao, "Software Defined Traffic Measurement with OpenSketch." in *Proc. Usenix Symposium on Networked Systems Design and Implementation (NSDI)*, vol. 13, 2013, pp. 29–42.

[70] K. Bu, X. Wen, B. Yang, Y. Chen, L. E. Li, and X. Chen, "Is every flow on the right track?: Inspect SDN forwarding with RuleScope," in *Proc. IEEE INFOCOM*, April 2016, pp. 1–9.

[71] A. Gupta and et al., "SDX: A Software Defined Internet Exchange," in *Proc. ACM SIGCOMM*, 2014, pp. 551–562.